Introduction

This module treats the problem of practical approaches to estimating the relative frequencies of a binary source. In higher order models, there may be many contexts. Each context is treated independently, however, so we can assume a single probability distribution for the adapter technique that estimates the relative frequencies.

Adaptation function $\alpha$ is a finite state machine. Simple adaptation employs count ratios (or scaled counts), and several count-ratio methods are discussed. The notion of speed of adaptation [4], or adaptation inertia, is presented. A section describes the concept of index-based adaptation, via a class of counting and forgetting techniques. An adaptive Golomb code [6] is designed with a wandering heuristic.

7.1 Dynamic Adaptation of Codes for Non-binary data

Approaches that learn the probability distribution of a data file while compressing the data are called one-pass approaches because they compress the data as they learn the distribution. Most such approaches, in one way or another, use counting techniques on the so-far encoded data in a way that results in shorter code length increases of the code string for the more frequently seen data. (An approach that does no counting and achieves the same end is called either LZ1 or LZ77, because it is based on a 1977 paper by Ziv and Lempel.)

Most dynamic adaptation techniques begin compressing a data file under an assumption of the initial probability of each symbol. The mathematical term prior means the initial assumption and is a technical term for “guesstimate”. Counting techniques then refine that estimate. Often the initial guess is the uniform distribution (all sample points are equally likely).

Another one-pass approach does “incremental look-ahead” that treats the file as a sequence of independent blocks. The algorithm counts the symbols in the next block, and then places a code at the beginning of the block that tells the decoder which code to use, and then encodes the block under the so-defined code.

7.1.1 Adaptive Huffman codes

Faller describes a one-pass adaptive Huffman coding technique for $M$-ary (many-symbol) alphabets in a conference paper [2] based on his Master’s thesis at the Federal University of Rio de Janeiro. The adaptive Huffman coding approach uses counting techniques and tree structures to dynamically generate a Huffman code tree of integer-length codewords. Each node of the tree is associated with a counter.

In his paper, Gallager [4] also describes the adaptive Huffman code. Faller’s algorithm employed the sibling property but Gallego coined the term and used the property for many new results.

Gallager used a “forgetting factor” in his adaptive Huffman code. The forgetting factor governs the speed of adaptation. Gallager suggests adjusting the counts through multiplication by a factor between 0 and 1 (the value of the “forgetting factor”) on a periodic basis (after a given number of events since the last adjustment, hence “total count”-based), and notes the speed of adaptation is determined by these parameters (the forgetting factor and the value of the total count). The higher the adaptation speed, the lower the adaptation inertia.

7.1.2 Context models and binary decomposition of larger alphabets

Although we treat binary (two-symbol) events, the counting concepts of the $M$-ary (three or more) symbol alphabets apply. Moreover, the technique of alphabet decomposition, allows the application of binary adaptation to $M$-ary alphabets. Decomposition makes binary alphabets and adaptation attractive because Shannon’s notion of entropy provides that the binary decomposition of any $M$-ary alphabet does not change the entropy value.

Early binary adaptation techniques [8, 7], were developed as universal parts for building and then combining with a coder and a context model designed for each particular application. Other adaptors appear in [12, 11, 10].

With binary adapters, a multiple-outcome event is transformed to a decision tree of binary events. Each leaf of the binary tree represents one of the $M$-ary symbols. The path from the root to the leave is a sequence of binary decisions whose outcomes are en-
coded. When the same path is decoded, the event coded was the symbol associated with the leaf.

Binary probability estimators are designed to provide a binary arithmetic code with the coding parameters (equivalent to a set of codewords for Huffman coding) in the form needed for that coder to encode (or decode) the next binary symbol.

### 7.2 Adaptation to Statistics

The goal for a binary adaptation algorithm \( \alpha \) (Greek letter for the first letter of the word “adapter”) is to determine the binary probability distribution for the next symbol or event of a binary sequence. The adapter represents engineering tradeoffs related to compression performance, the coder interface, the execution time per event, and the cost.

In a general compression setting, adapters determine \( P(0|z,s[z]) \), i.e., \( \alpha \) dynamically determines an estimated probability \( p(LPS)_{est} \) for binary event \( b \) in context \( z \), having seen a subsequence \( s[z] \) of binary events under context \( z \) up to the current point in time. The binary sequence \( s[z] \) is the subsequence of the so far seen sequence \( s \) obtained by selecting only those binary symbols labeled by context \( z \).

Alternatively, if we know the value of the less probable of the two binary symbols, then let \( p(LPS) \) denote the probability of the less probable symbol (denoted LPS) for the binary string whose relative frequencies are to be estimated. Let \( p(LPS)_{est} \) denote our estimate of the relative frequency of the less probable symbol. The adapter’s task is to determine the current value \( p(LPS)_{est} \), as the estimate of \( p(LPS) \), from the past (i.e., without looking into the future).

#### 7.2.1 Computational models for finite adaptation algorithms

Adapter algorithms \( \alpha \) store a state; i.e., have memory. Practical algorithms are finite so adaptation to the statistics is by a finite state machine (FSM). The adapter examines the next symbol (if the encoder) or the most recently decoded symbol (if the decoder), and adjusts or not the adapter state. The state determines the current estimate

The binary adapter produces an estimated value \( p(LPS)_{est} \) (for what is about to be seen) that should approximate the value \( p(LPS) \) (for what actually will be seen). An adaptation parameter can include (or have pre-stored with the parameter) the index component of a coding parameter for coding purposes. Binary coders may conveniently employ a coding parameter by quantizing the set of all distributions into a finite set, each member corresponding to an index. Such parameterized coders use a relatively small set of probability estimates.

Each coding parameter or index represents a skewness estimate that covers a quantization range of estimates. For the binary alphabet, a simple skewness estimate is \(- \log_2 p(LPS)\). This skewness measure takes on integer values 1, 2, 3, 4, etc., for the respective values of \( p(LPS) \) of \( \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \) etc.

Suppose we use the logarithm to base 2 for the quantization ranges, and value \( \log_2 K \), where \( K = 1, 2, \) etc., as the representative value of \( p(LPS)_{est} \) for values of \( p(LPS) \) that fall within its range. We are clearly interested in knowing for what values of \( p(LPS) \) we should use \( \frac{1}{4} \) instead of \( \frac{1}{8} \), for values lying between \( \frac{1}{8} \) and \( \frac{1}{16} \), and the high side of \( \frac{1}{16} \). The answer turns out that above \( p(LPS) \) of 0.368 we should use \( \frac{1}{16} \). Also, on the low side of \( \frac{1}{16} \) we should use \( \frac{1}{8} \) for values of \( p(LPS) \) as low as about 0.180, below which value \( \frac{1}{32} \) becomes more efficient. We call 0.368 and 0.180 the “breakpoints” of the quantization range for index 2, i.e., for using \( p(LPS)_{est} \) of \( \frac{1}{4} \). At the upper break point \( p(LPS) = 0.368 \), \( p(LPS)_{est} \) of \( \frac{1}{4} \) and \( \frac{1}{8} \) offer the same inefficiency, and at the lower break point \( p(LPS) = 0.180 \), \( p(LPS)_{est} \) of \( \frac{1}{8} \) and \( \frac{1}{32} \) offer the same inefficiency.

If \( p(LPS)_{est} \) estimates the probability then \(- \log_2(p(LPS)_{est})\) is the skewness of the estimate; the smaller \( p(LPS)_{est} \), the more skewed the distribution. Index \( K \) points to a table entry that that characterizes the estimate. An internal state index of adapter \( \alpha \) can do a table lookup to find whatever values are needed by any given binary arithmetic coder.

#### 7.2.2 Customizing the code to the data file

The static approach to encoding the code employs a fixed code designed for data files in a particular class. For example, the early bilevel image compression algorithm for facsimile (fax machines) has a fixed code table designed for printed documents. If the document does not belong to the class to which the code tables were designed then the result of “compression” is a coded data file that exceeds the size of the original data. Expanding the data with a compression algorithm does not win management approval. When scanned gray-level or half-toned data are presented to the so-called Group 3 facsimile algorithm of the CCITT, the data typically expands by 50%, i.e., the output file is 150% the size of the original data.

The advantage of compression techniques that use a fixed coding technique is simplicity and absence of a header. However, the danger with any compression technique that uses a fixed coding technique is suboptimal compression. In some cases, there may be several fixed codes available, say eight. In this case a simple 3-bit field in the header can describe the code. The added complexity is that usually the file needs to be read first before deciding which code is best.

Two uses of symbol counting to customize the code on a file basis are (1) the one-pass (adaptive, dynamic, on-line, on-the-fly) estimation algorithms, and (2) the
two-pass (semi-static, off-line) estimation algorithms.

In the simplest one-pass approach, the Laplace's estimate is used: the frequency count of each symbol is initialized to one, and the total frequency is initialized to the number of symbols in the alphabet. The symbol frequency count of the just-encoded (or just-decoded) symbol is incremented by one. The probability of the next symbol to be encoded is its relative frequency: the ratio of its frequency count to the total symbol count. Since arithmetic coding uses the relative frequencies directly, the one-pass method is more easily applied to arithmetic coding than to Huffman coding.

The two-pass count technique has a first pass that increments each symbol's frequency count after reading that symbol. As in the one-pass case, the first pass of the two-pass approach does the statistics-gathering. The statistics, coding parameters for arithmetic coding, or codewords (or symbol counts) for Huffman coding, are transmitted to the decoder as a preamble or header to the code string. Next a second (coding) pass over the data file uses the resulting estimated probabilities to encode the data.

7.2.3 One-pass binary count ratios

With the advent of binary arithmetic codes coupled with binary adaptation, early count-ratio (scaled-count) binary adaptation techniques appeared. In the cumulative count technique (Laplace’s estimate or Laplace’s estimator) the counts are accumulated on-the-fly as the file is read and are used for compression on the same pass.

In [12, 11], the cumulative count ratios adaptively determine the probabilities for 0 and 1 for direct and instant use in a binary arithmetic code. In other words, the binary probability range was not quantized.

For the binary source, one may count the total number of symbols (0 or 1) seen plus count the instances of the symbol 0. The count of symbol 1 is the difference between the total count and the count of symbol 0. One could also count each symbol and sum them to obtain the total count.

Another method is to maintain the binary value of the LPS and the total count. The value of the MPS count is the difference between the total count and the LPS count. Alternatively one can maintain the total and MPS count, or the LPS and MPS counts.

7.3 Binary Adapter Performance on Non-stationary Binary Sequences

Data compression experience indicates that data such as bilevel images contain context dependent binary sequences $s[z]$ with nonstationary jumps. Thus adaptation speed, or inertia to change, is an important practical issue for adjusting the estimate in accordance with nonstationary jumps. For nonstationary data, the ideal code length or per-bit length calculation serves as a type of lower upper bound to compression rather than as a lower bound.

If one knows a data sequence is non-stationary, and has an estimation algorithm that follows the changes in probability, then one can achieve “sub-entropy” performance. In fact, the best way one knows the data is non-stationary is when the code string length is shorter than the stationarity assumption would permit.

The notion of entropy itself depends on the cumulative count technique applied to an infinite sequence with no forgetting; which in turn provides the same answer as with the stationarity assumption. For stationary sequences the one-pass cumulative count (Laplace’s estimator) is an excellent approach to adaptation coupled with arithmetic coding. Typically the encoding using the one-pass cumulative counts achieve the compression performance as determined by the Enumerative Length (closed form) calculation EL.

7.4 Adaptation Speed

7.4.1 On the size of the total count number

Here we treat adaptation inertia based on counting techniques, as relate the inertia for binary estimators to the value of the probability of the less probable symbol and the value of the total symbol count used by the estimator.

Consider an example of the uniform distribution, with the binary distribution parameter $p(LPS)$ near one half. Let the total count (which is the count ratio denominator) be 1,000 and the LPS count be 400. This example will show that the size used for the total count determines how quickly the scaled count approaches can react to a non-stationary jump in the statistics of a binary source.

Clearly, with the denominator (the total count) at 1,000, and given a nonstationary jump in the statistics such that the value of $p(LPS)$ becomes 1/100, then adapting to this nonstationary change is relatively slow. For example, after 1,000 symbols at this new value of $p(LPS)$ the expected count ratio is 500/2000 or 1/4. This is not even close to 1/100. Our example demonstrates that at a current estimate of $p(LPS)$ of 1/2, which represents a skew number of 1, that a total count of 1,000 symbols represents a high inertia to change.

On the other hand, suppose $p(LPS)$ is 1/500, which is a skew number close to 9. Now a total symbol count of 1,000 for the denominator is more reasonable, and in fact may even be too small.

In summary, much less than 1000 recent samples justify an estimate in the 0.5 to 0.25 range, and in order to represent $q_{MPS}$ of 0.001 (skewness of about 10) as a count ratio we need more than 1000 recent samples. Our discussion suggests that the smaller the probability of the LPS, the longer the sequence of past history that is needed for making the estimate, and motivates the description of the next algorithm.
Thus, a forgetting scheme based on total counts has a non-uniform adaptation speed problem: adaptation is slower for lower-skewed data.

### 7.4.2 Uniform or non-uniform inertia

Another issue with adaptation speed concerns the relative weight of each new count with respect to the total count or the denominator of the count ratio. If the total count varies from 0.5N to N, then as the denominator approaches N the newer counts weigh half as much as those that occur just following the reduction of the denominator to 0.5N. A more “constant” inertia approach is to make smaller reductions at more frequent intervals. For example, controlling the denominator between 0.75N and N, or between 0.875N and N, serves to narrow the range of values of the scaled count denominator, provide more frequent scaling, and thus provide a more uniform inertia.

### 7.5 The Skewcount (SSS-adap) family

In this section we describe a scaled-count approach to binary probability estimation, and then provide two examples of an algorithm called LPS3:5.1. In the algorithm name, we consider the first value 3 as the value assigned to variable $LPS_{max}$, the second value 5 as $LPS_{max}$, and third value 1 as the count increment $Inc$.

In dealing with binary probabilities, there is an implementation advantage (in hardware at least), to assign a truth variable $LPSVal$ to indicate whether binary digit 0 or 1 is the less probable symbols, denoted LPS. $LPSVal$ is 0 if $p(0) < 0.5$, and $LPSVal$ is 1 if $p(1) < 0.5$. The initial state of LPSVal is arbitrary if we initially assume $p(0) = p(1) = 0.5$. The first symbol seen, 0 or 1, typically defines the more probable symbol, denoted the MPS. Once LPSVal is defined, if $p(LPSVal)$ reaches a state where $p(LPS) = 0.5$, we assume some “inertia” and do not change LPSVal. Such a policy can reduce needless changes to the state of LPSVal. Since we have a binary algebra, only two counts are needed. We choose the count of the LPS, denoted $LPS_{ct}$, and the count total, denoted $TOT_{ct}$, as the count-pair to be maintained.

Algorithm SSS-adap constrains the LPS count to values 3, 4, and 5. When the LPS count reaches 6, the LPS count is halved to 3, and the count total is halved as well. The algorithm LPS3:5.1 is so-called because the LPS count range is 3 to 5, and the count increment is 1. This algorithm is described in [8]. With value $LPS_{ct}$ as the current count for the LPS symbol, and $TOT_{ct}$ as the total count, their current count ratio is the value $PLPS$, the probability of the less probable symbol. The probability estimate $pLPS$ is: $pLPS = \frac{LPS_{ct}}{TOT_{ct}}$.

We describe LPS3:5.1 by means of a program snippet. We assume there is some function that is passed the symbol (0 or 1), value of $LPS_{ct}$ and value of $TOT_{ct}$, and is returned the new values of $LPS_{ct}$ and $TOT_{ct}$. Note that the encoding (decoding) of each binary symbol must occur before count updating. This is necessary for the decoder (who will otherwise not know the symbol value) and also the encoder (who will otherwise use counts that are out-of-step with the decoder). What follows is one possible version of an algorithm in the class LPS3:5:1.

```c
Version 1. LPS3:5:1
/* Update the counts for LPS3:5:1 */
TOTct++; // Inc LPS count later */
if (TOTct >= remnormval) {
    TOTct >>= 1; /* Halve TOTct */
    LPSct >>= 1; /* Halve LPSct */
    /* Re-scaling yield too small LPSct? */
    while (LPSct < 3) { 
        LPSct++;
        TOTct++;
    }
}

if (symbol == LPSval) LPSct++; /* "LPS up" */
/* Now re-scale based on LPSct. */
if (LPSct >= 6) {
    LPSct = 3;
    // TOTct = TOTct/2;
    /* Re-scaling may have changed LPSval */
    if ((2 * LPSct) > TOTct) {
        LPSval = 1 - LPSval; /* Swap LPSval */
        TOTct = 2 * LPSct;
    }
}
```

There are some implicit “rules” in the above:

1. $TOT_{ct} <= \text{renormval}$
2. $LPS_{min} <= LPS_{ct}$
3. $tt LPS_{ct} <= LPS_{max}$
4. $LPS_{ct}/TOT_{ct} <= 0.5$

The motivation for rule 1 is that the coder has a limit on how small p(LPS) can be. Hardware encoders have some maximum number of bits of precision, for economical reasons. Rule 1 is intended to prevent p(LPS) from requiring too many leading 0s in the binary fraction representation because the larger the denominator, the smaller the binary fraction. For example, in the paper describing SSS-adap [8], the smallest allowable probability was either $2^{-15}$ or $2^{-14}$, so there was no point for $TOT_{ct}$ to exceed 64K. In fact, using 16-bit integer unsigned arithmetic, one runs into technical problems with counts that become too large.

Rule 2 state that $LPS_{ct}$ will not fall below $LPS_{min}$. Similarly, Rule 3 states the $LPS_{ct}$ does not exceed $LPS_{max}$. Rule 4 ensures that $p(LPS) <= 0.5$. However, suppose we have $LPS_{ct}$ of 5 and $TOT_{ct}$ of 10,
and we get an LPS. In the above code, the new TOTct is 11/2 = 5 and new LPSct is 3.

Many of the alternatives for rules 2 and 3 will make little difference in the compression of most data. The general question of the effect on compression of these minor or “arbitrary” choices has not been fully investigated, to our knowledge. The main point here is that the encoder and decoder must keep “in step”, and use the same probability estimator.

The typical scenario is that most of the time an MPS occurs, and so the MPS operation is performed: value TOTct is incremented and tested against TOTmax. The test has two outcomes.

Two Cases (TOT1, TOT2) after TOTct incremented
1. TOTct does not exceed TOTmax.
2. TOTct exceeds TOTmax.

For Case TOT1 above, no action need be taken. However, for Case Tot2, TOTct is either overflowing the number of bit positions in the register, or for some encoding or decoding reason the limit is reached. Two solutions come immediately to mind. First, one can forbid the incrementing of the variable TOTct beyond TOTmax, and allow TOTct to remain at TOTMax. This first alternative is called “clamping” TOTct at TOTmax. Second, one may choose to scale back TOTmax, and possibly scaling back the value of LPSct as well.

The scaling or forgetting factor based on TOTmax can be different from that which occurs when LP-Sct exceeds LPSmax. For example when TOTmax is reached we may wish to simply scale back TOTct by 3/4 (leaving LPSct alone), while scaling back both TOTct and LPSct by 1/2 when LPSmax is reached. In the example implementation of LPS3:5:1, when TOTmax was reached, LPSct is scaled back before LPSct is incremented. Alternatively, given that TOTmax is reached and the symbol is LPSval, LPSval could be scaled back after it is incremented.

Four Cases for LPS (LPS1, LPS2, …, LPS4) after TOTct incremented
1. Scaling back TOTct and LPSct leaves LPSct less than LPSmin (Rule 2) (handled together with incrementing TOTct).
2. LPSct has been incremented and LPSct exceeds LPSmax (Rule 3).
   • Action: Halve LPSct and TOTct.
3. LPSct has been incremented and LPSct × 2 > TOTct. (Rule 4)
   • Action: LPSct has been incremented, is within its range, and LPSct × 2 ≤ TOTct.
   If Case LPS1 occurs for the LPS, we arranged to treat this case first before learning whether the symbol is MPS or MPS, because we bring LPSmin back to LPSmin only if needed. For cases LPS2, and LPS3, we need to do Case 2 before we can test against TOTct (case 3). Above, we test the LPS count against the respective rules, and take the appropriate action to cause values LPSct and/or LPSval to comply with the appropriate rule. Case 2 or Case 3 or Case 4 occurs, then there are alternatives to consider.

Further Comment on the LPS Val Test
Two strategies exist for the “Swap LPSval” test, depending on whether we answer the following question a “Yes”, or “No”: should we perform the test 2 × LPSct > TOTct after every MPS operation? We have gone with “simplicity”, which suggests we can perform the “Swap LPSval” test only after halving.

Comment on binary probabilities near 1/2
In dealing with the binary distribution, folk wisdom suggests that one achieves good compression even when the probability estimate is a “little off”. The “folk wisdom” is particularly good when the distribution is highly skewed, i.e., when p(LPS) is very small. However, when the distribution is near uniform, i.e., when p(0) ≈ p(1) ≈ 1/2, the folk wisdom breaks down. In the vicinity of the uniform distribution, the danger is that the compression algorithm may expand the data.

The binary entropy curve is flat in the vicinity of 1/2. Suppose the actual probabilities are 0.48 and 0.52. The entropy can be calculated from the ideal length of a sequence of 25 symbols: 12 0s and 13 1s. IL = 25 log2 25 - 13 log2 13 - 12 log2 12 = +116.10 - 48.11 - 43.02 = 24.97. So the entropy is 0.9988. So if, by good fortune, we estimate the probabilities exactly, the resulting compressed file is 0.1%, (or smaller by 0.001, or 0.999 of its original size).

The compression gain is hardly worth it, because small errors in the probability estimation can be disastrous: much worse than just assuming that each of the binary events has probability 0.5. Suppose for example, the 12 and 13 were reversed. Now the self-information values subtracted from 116.10 become 46.61 and 44.4, or 25.09, or 1.0036 bits/symbol. Instead of 0.1% compression, the data is expanded by 0.35%.

With the LPS3:5:1 estimator, the “granularity” of the estimate is quite coarse near the uniform distribution. It is known that these scaled count estimators work much better if the data is more highly skewed (i.e., if p(MPS) is large and p(LPS) is small).

Alternatively, since the problem is the coarse grain of the integer count ratio, one may wish to increase
the number of values available to the numerator and
denominator whose ratio determines the estimate.

**Example 1.** LPS3:5:1. Swapping LPS.val.

Consider the following 10-symbol string:

```

<table>
<thead>
<tr>
<th>Pos'n</th>
<th>1 2 3 4 5 6 7 8 9 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>0 0 0 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>
```

Let symbol 0 be the initial LPS (less probable), let
the initial counts be 5 for the LPS and 10 for the total
count. In this way, if the first symbol is the LPS, we
may wish to revise our estimate as to the LPS.

**Position 1.** The symbol is 0, the LPS, we increment
LPSct 5 to 6, and TOTct to 11. We halve so LPSct =
3, TOTct = 5 and discover 2 x 3 > 5 and swap roles.
New TOTct = 2 x LPSct = 6, LPSct remains at 3.
LPSval is 1.

**Position 2.** The symbol 0 is now MPS, so TOTct be-
comes 7.

**Position 3.** The symbol 0 is MPS, so TOTct becomes
8.

**Position 4.** The symbol 1 is LPS, so LPSct = 4 and
TOTct = 9.

**Position 5.** The symbol 1 is LPS, so LPSct = 5 and
TOTct = 10.

**Position 6.** The symbol 1 is LPS, so LPSct = 6 and
TOTct = 11. We halve so LPSct = 3, TOTct = 5 and
discover 2 x 3 > 5 and swap roles. New TOTct = 2 x
LPSct = 6, LPSct remains at 3. LPSval is 0.

Positions 7, 8, 9, and 10: TOTct moves to 7, 8, 9, and
10.

**Summary of Example 1**

<table>
<thead>
<tr>
<th>Curtn</th>
<th>Final</th>
<th>Final</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Posn</td>
<td>Symb</td>
<td>LPSval</td>
<td>LPSct</td>
</tr>
<tr>
<td></td>
<td>-----</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>Init</td>
<td>-</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>etc.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

So LPSct becomes 1 + 2 = 3 (LPSmin) as TOTct
becomes 63 + 2 = 65.

Variations on the theme LPSmin/hinc exist. For ex-
ample, in [9], family member LPS1:1:1 is used. When
the LPS count reaches 2, the denominator (LPS count)
is halved. Skewcount retains the LPS count at 1: when
the LPS count of 1 is incremented to 2, both counts
are halved so the LPS count returns to 1. Skewcount is
intended for highly skewed and highly non-stationary
binary strings. It performs very poorly on the uniform
distribution, and “wanders” too much for skewed dis-
tributions that are stationary. LPS1:1:1 or LPS3:5:1
always cause an index change at least each third LPS
event (for LPS3:5:1).

### 7.6 The Goladap Adapter-Coder

An interesting adaptive single-context binary (or runlength) code of the wandering type is called Go-
ladap [6]. This adaptive runlength code employs the
adaptation heuristic of the previous section, but com-
binds adaptation with an operation inherent to the
Golomb coding algorithm. Golomb [5] defines parame-
ter \( m \) (or Golomb number \( m \)) such that the probability
of \( m \) successive MPS symbols is 0.5, i.e., \( p_{\text{MPS}}^m \approx 0.5 \).

The Golomb code for a particular Golomb number
\( m \) represents the runlength to be encoded as two parts: first the unary (pronounced “you-na-ree”) number (base 1 or “stone-age binary”) which is the integer quotient of \( L \div m \), and second the integer
remainder of \( L \mod m \). If \( L \div m \) results in integer \( i \), then its unary representation consists of \( i \) is followed
by a single (terminating) 0. For the unary system,
there needs to be some way of ending the unary string.

In [1], only the Golomb numbers \( m \) that are pow-
ers of 2, i.e., \( m = 2^k \), are used. Thus, following the
delimiter bit that ends a unary number, there are
exactly \( k \) bits for the remainder part 0,1,...,\( 2^k - 1 \).
Counter value \( m \) is initialized to 0, then counted up
for each MPS, and a 1 shifted onto the codestring when-
ever the counter rolls over. For the LPS outcome, a 0
shifted onto the codestring terminates the unary num-
ber. Then the \( k \)-bit counter value is shifted onto the
codestring. Goladap also uses values of \( m \) that corre-
spond to a power of 2, and increases the adaptation
index (Golomb number) when a stone-age 1 is shifted
onto the codestring.

If the incoming binary sequence statistics are
matched to Golomb number \( m \), then the modulo \( m \)
counter reaches count \( m \) with probability approxi-
mately one-half. Goladap uses the balance point of the
two-sided significance test, which is reached when
the first stone-age 1 occurs. This balance point is the
technical basis for the proof of optimality of the code
in [3]. For a given binary distribution parameter \( q_{\text{MPS}} \) for
the incoming stationary binary sequence, approxi-
mately half the Golomb codewords should have no
stone-age 1, and the other half should. In Goladap,
the balance point decides the one-sided test which always rejects the null hypothesis, and so the adaptation inertia is very high. Goladap is clearly a wandering adaptation algorithm.

The mechanism of the coding part of the Goladap adapter-coder is described in [1, 3]. During the encoding process, when the modulo \( m \) counter rolls over and a new state-age 1 is shifted onto the codeword, the adapter changes the index \( k \) to the next higher permissible value. For the LPS that ends the run, Goladap decrements the current value of \( k \). The Goladap coder was developed after Skewcount, and employs the balance point very much as the binary adaptation heuristic described in connection with Skewcount. If more than the expected MPS events occur before the termination of the run, then index \( k \) is increased to \( k + 1 \) and then the LPS event decreases the index back to the previous value of the index. On the other hand, if the LPS decreases \( k \) first, then at least one \( m \)-counter rollover at the new index \( k - 1 \) is expected, which brings the index back up to \( k \).

Goladap is a combination of Skewcount and the Golomb code. Goladap also brings to the adaptation process the inherent modulo \( m \) counting that the Golomb coding procedure performs anyway. The index-change decision, or adaptation, is triggered only when the code length is increased, either by the \( m \)-th MPS event or the LPS event that terminates the run. In retrospect, the LPS outcome triggers the less skewness one-sided test, and the \( m \)-counter rollover triggers the more skewness one-sided test.

The Goladap runlength adapter-coder was tested against the test files employed in [8] with a 2-context first-order binary model (a white run followed by a black run, etc.). For single-context runlength codes, one may switch contexts after the LPS event. Goladap performs approximately 15% better than an adaptive runlength code due to Weber [13], but Weber did slightly better on digital halftone data.

7.7 Practical Issues in One-pass Multiple Context Situations

A count ratio implies division. However, table lookups replace division at the cost of an approximation. The leading bits of the divisor and dividend, and an integer representing the difference in the most significant bit of each, serve as as an address for a lookup that delivers the quotient approximation.

The multiple context case has a sparse context problem. Too many contexts for a given data file may distribute the symbol counts unevenly distributed such that many contexts have too few symbol counts to adequately approximate the statistics.

A tool based on the difference between NlogN and Logfac can detect the sparse context problem as follows. The calculation NlogN creates too few bits for a sparse context that has many symbols of zero count.

Symbols of 0 count get 0 code space, which reduces the alphabet size. In fact, if only one symbol occurs, the IL for that context is 0.

In contrast, for sparse contexts with few counts Logfac gives a “uniform distribution” result: almost no compression! Why? There are too few counts to change much from the initial uniform distribution. For low-frequency contexts, Logfac reports little compression and NlogN reports excellent compression. The more severe the sparse context problem, the wider the difference between NlogN and Logfac. On the other hand, with well populated contexts, the NlogN and Logfac measures may differ by less than 1%.

Another practical issue for one-pass probability estimators is the problem of quickly learning the binary distribution is highly skewed or not. The general idea is called a “fast attack” or “early attack”. For scaled count estimators, the fast attack takes the form of using count increments larger than 1 for the first several symbols. For example, in the binary case, we can assume the first symbol seen is the More Probable Symbol, and add a large increment (say 16) to the total count. For the second symbol, a lesser increment (say 8) can be added. When two or three of the less probable symbols have been seen then the count increments can be reduced to unity (count increments of 1).

7.8 Summary

The statistical significance approach (SS-adap) of [8], and the multilevel adapter for nonstationary sources [10] are powerful approaches. The wandering binary adaptation algorithms that use deterministic decisions have been described, and were discovered through a significance testing approach. Adaptation to a higher skewness is counter-balanced by the adaptation inertia in the other direction so \( p(\text{LPS})_{\text{est}} \) does not wander far from the actual \( p(\text{LPS}) \). Higher inertia wanders less but reacts more slowly to nonstationary jumps in \( p(\text{LPS})_{\text{LPS}} \).

References


Additional references

This module is based on the following reports:

